디지털 논리회로

최종보고서

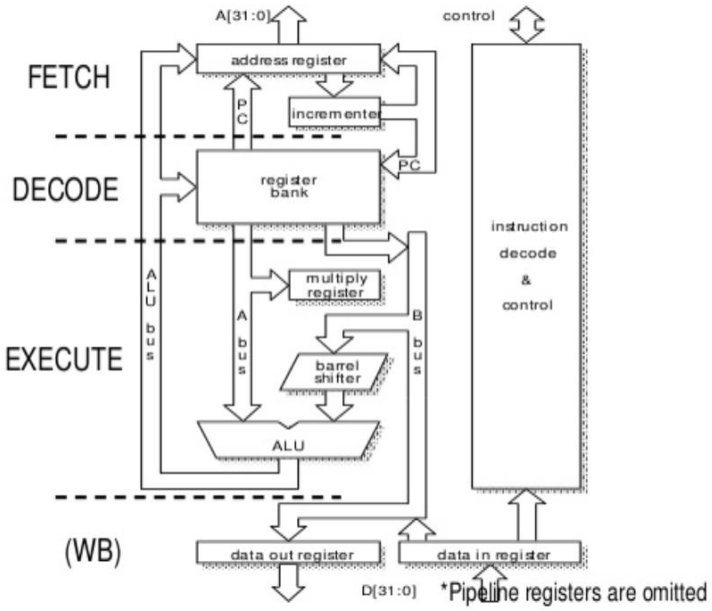
제출일	2024.06.11	학교	숭실대학교
과목	디지털 논리회로	조	1조
담당교수	박재근	조원	김형주 20182140 김동희 20211060 정민지 20211099

목차

1. 프로젝트 (CPU 설계)

1.1 프로젝트 목적

본 프로젝트는 다음 그림에 해당하는 것과 같이, Fetch-Decode-Execute 의 3단계 파이프라인을 갖는 미니 CPU를 설계하고자 한다. Fetch 단계에서 명령어를 메모리에서 인출하고, Decode 단계에서는 명령어를 해석하고, execute 단계에서는 명령어를 실행하도록 설계하고자 한다.



'미니 CPU 설계' 라는 프로젝트에 알맞게 간단한 산술 연산을 하는 CPU를 설계하고자 했다. 특히 수열의 합인 $\sum_{i=S}^{N} i$ (N>=S), 즉 S+(S+1)+(S+2)+ … +(N-1)+N 을 구하는 동작을 하도록 설계하는 것을 목표로 프로젝트를 진행한다.

1.2 명령어 정리

ㄱ. 명령어 format 결정

본 프로젝트에서 설계한 CPU는 다음과 같은 명령어 format을 따른다.

Opcode(4bit)	S(12 bit)

상위 4bit는opcode로 사용되어 명령어의 종류를 지정하였다. 뒤의 12bit의 S 부분은 주소를 나타내는 부문으로 수행할 명령어의 데이터가 있는 곳의 주소를 가르킨다.

ㄴ. 사용한 명령어의 종류와 기능

: $\sum_{i=S}^{N} i$ (N>=S) 연산을 위해 다음과 같은 명령어를 정의한다.

주소	instruction	동작
1	LDA S	acc := mem[S]
2	STO sum	mem[sum] := acc(S)
3	STO i	mem[i] := acc(S)
4	SUB N	acc(S) := acc(S) - mem[N]
5	JNE loop1	if acc != 0 pc := loop1
6	STP	stop
7, loop1	LDA i	acc := mem[i]
8	ADD v1	acc := acc(i)+mem[v1]
9	STO i	mem[i] := acc(i+1)
10	ADD sum	acc(sum) := acc(i) + mem[sum]
11	STO sum	mem[sum] := acc(sum)
12	LDA i	acc := mem[i]
13	SUB N	acc := acc(i)-mem[N]
14	JNE loop1	if acc != 0 pc := loop1
15	STP	stop

1.3 모듈 상세 설명

- **subtotalCPU**

```
module subtotalCPU(in_data, rst_n, clk,
          out_data, out_address, memrq, rnw);

    input [15:0]in_data;
    input rst_n, clk;
    output [15:0] out_data;
    output [11:0] out_address;
    output memrq, rnw;
    wire acc_ce, acc_oe, acc_15, accz;
    wire a_sel, b_sel, decode_ce, fetch_ce;
    wire [2:0] execute_fs;
    wire [3:0] opcode;
    wire [11:0] addr_from_decode, addr_from_fetch;
    wire [15:0] data_execute, data_mux, data_acc_to_execute;

    ACC u_acc(.in_data(data_execute), .acc_ce(acc_ce), .acc_oe(acc_oe), .clk(clk),
          .out_to_bus(out_data), .out_data(data_acc_to_execute), .acc_15(acc_15), .accz(accz));
    execute u_execute(.in_A(data_acc_to_execute), .in_B(data_mux), .execute_fs(execute_fs),
          .out(data_execute));
    control_logic u_control_logic(.in_opcode(opcode), .acc_15(acc_15), .accz(accz), .clk(clk), .rst_n(rst_n),
          .a_sel(a_sel), .b_sel(b_sel), .decode_ce(decode_ce), .fetch_ce(fetch_ce), .acc_ce(acc_ce), .acc_oe(acc_oe), .execute_fs(execute_fs),
          .rnw(rnw), .memrq(memrq));
    fetch u_fetch(.in_instruction(in_data), .fetch_ce(fetch_ce), .clk(clk),
          .out_opcode(opcode), .out_address(addr_from_fetch));
    mux_2to1 #(.WORD(12)) u_mux_a(.i0(addr_from_decode), .i1(addr_from_fetch), .sel(a_sel),
              .out(out_address));
    mux_2to1 u_mux_b(.i0({4'b0000, out_address}), .i1(in_data), .sel(b_sel),
              .out(data_mux));
    decode u_decode(.in_address(data_execute), .decode_ce(decode_ce), .clk(clk),
          .out_address(addr_from_decode));

endmodule
```

- **input [15:0]in_data** : 16비트의 입력신호
- **input rst_n, clk** : 1비트의 입력 신호
- **output [15:0] out_data;** : 16비트의 출력 신호
- **output [11:0] out_address;** : 12비트의 출력 신호
- **output memrq, rnw**; : 1비트의 출력 신호

- 내부적으로 사용되는 *wire* 선언
- ACC, execute, control_logic, fetch, mux, decode 모듈의 인스턴스 생성

● Fetch

: data에서 opcode와 address를 분리하여 opcode는 control logic에 전달하고, address는 memory로 전달하여 해당 주소에 저장 되어있는 값을 참조하거나 명령어의 주소를 execute에 전달하여 +1만큼 증가시키고 최종적으로 Decode에게 전달한다. Control logic에서 *fetch_ce* 신호를 1로 받을 때만 clk의 positive edge에서 값을 저장한다.

```
1  module fetch(in_instruction, fetch_ce, clk,
2            out_opcode, out_address);
3
4      input [15:0]in_instruction;
5      input fetch_ce, clk;
6      output reg [3:0] out_opcode;
7      output reg [11:0] out_address;
8
9      always @(posedge clk)
10     if (fetch_ce)
11         begin
12             out_opcode  <= in_instruction[15:12];
13             out_address <= in_instruction[11:0];
14         end
15
16 endmodule
17
```

- **always @(posedge clk)** : 클록의 상승 에지에서 동작
- **if (fetch_ce)** : 'fetch_ce' 활성화되면 동작.
- **out_opcode <= in_instruction[15:12];** : 'in_instruction'의 상위 4비트를 'out_opcode'에 할당
- **out_address <= in_instruction[11:0];** : 'in_instruction'의 하위 12비트를 'out_address'에 할당

- **ACC**

  : accumulator 누산기 라고도 하며, execute에서 출력된 연산 결과를 저장하고, memory로 전달하는 역할을 한다. Control logic에 의해 주어진 신호 *acc_oe, acc_ce* 에 따라 어느 방향으로 출력을 전달할지 결정하며 *acc_15* 그리고 *acc_z* 신호를 추출하여 control logic에 전달한다.

```
1  module ACC(in_data, acc_ce, acc_oe, clk,
2             out_data, out_to_bus, acc_15, accz);
3
4      input [15:0] in_data;
5      input acc_ce, acc_oe, clk;
6      output reg [15:0] out_data, out_to_bus;
7      output reg acc_15, accz;
8
9      always @(posedge clk) begin
10         if (acc_ce)
11             out_data <= in_data;
12     end
13
14     always @(out_data) begin
15         acc_15 <= out_data[15];
16         accz   <= ~|out_data;
17     end
18
19     always @(*)
20         if (acc_oe)
21             out_to_bus <= out_data;
22         else
23             out_to_bus <= 16'b_zzzz_xxxx_zzzz_xxxx;
24
25 endmodule
26
```

- **if (acc_ce)**

  out_data <= in_data; : 'acc_ce'가 활성화되면 'in_data'를 'out_data'에 할당

- **always @(out_data) begin** : 'out_data'의 변화에 따라 동작

- **acc_15 <= out_data[15];** : 'out_data'의 15번째 비트를 'acc_15'에 할당

- **accz <= ~|out_data;** : 'out_data'가 모두 0이면 'accz'를 1로, 아니면 0으로 설정
- **if (acc_oe)**

  out_to_bus <= out_data; : 'acc_oe'가 활성화되면 'out_to_bus'에 'out_data'를 할당
- **else out_to_bus <= 16'b_zzzz_xxxx_zzzz_xxxx;** : 'out_to_bus'에 'out_data'가 할당되지 않으면 'out_to_bus'를 불확정 상태로 설정

● Decode

: 다음에 실행할 명령어의 주소를 가리키고 있는 레지스터이다. Program Counter라고하며, execute로부터 다음에 실행할 명령어의 주소를 전달받는다. Control logic에 의해 decode_ce신호가 1일 때, clk의 positive edge에서 값을 저장한다.

```
1  module decode(in_address, decode_ce, clk, out_address);
2
3      input [15:0] in_address;
4      input decode_ce, clk;
5      output reg [11:0] out_address;
6
7      always @(posedge clk)
8          if (decode_ce)
9              out_address <= in_address[11:0];
10
11 endmodule
12
```

- **if (decode_ce) out_address <= in_address[11:0];** : 'decode_ce'가 활성화되면 'out_address'에 'in_address'의 하위 12비트를 할당.

- **Execute**

  : 산술논리연산을 하는 모듈로 control logic에 의해 주어진 신호 execute_fs에 의해 input값을 더할지, 뺄지, +1증가시킬지 결정하여 출력한다.

```
1  module execute(in_A, in_B, execute_fs, out);
2
3      input [15:0] in_A, in_B;
4      input [2:0] execute_fs;
5      output reg [15:0] out;
6
7      always @(*) begin
8          case (execute_fs)
9              3'd0: out <= 16'b0;
10             3'd1: out <= in_A + in_B;
11             3'd2: out <= in_A - in_B;
12             3'd3: out <= in_B;
13             3'd4: out <= in_B + 1;
14             default: out <= 16'b_zzzz_zzzz_zzzz_zzz;
15         endcase
16     end
17
18 endmodule
19
```

- **always @(*) begin**

  case (execute_fs) : ' execute_fs '의 값에 따라 'out'의 값을 결정

- **3'd0: out <= 16'b0;** : execute_fs가 0일 때 'out'은 0으로 설정
- **3'd1: out <= in_A + in_B;** : execute_fs가 1일 때 'out'은 'in_A'와 'in_B'의 합으로 설정
- **3'd2: out <= in_A - in_B;** : execute_fs가 2일 때 'out'은 'in_A'에서 'in_B'를 뺀 값으로 설정
- **3'd3: out <= in_B;** : execute_fs가 3일 때 'out'은 'in_B'의 값으로 설정
- **3'd4: out <= in_B + 1;** : execute_fs가 4일 때 'out'은 'in_B'에 1을 더한 값으로 설정
- **default: out <= 16'b_zzzz_zzzz_zzzz_zzz;** : 그 외의 경우 'out'은 불확정 상태로 설정

- **Mux**

  : 2개의 입력 중 어느 입력을 출력할지 결정하며 control logic에 의해 제어된다.

```
1  module mux_2to1(i0, i1, sel, out);
2
3      parameter WORD = 16;
4      input [WORD-1:0] i0, i1;
5      input sel;
6      output reg [WORD-1:0] out;
7
8      always @(*)
9          case (sel)
10             1'b0: out <= i0;
11             1'b1: out <= i1;
12             default: out <= 16'bzzzz_zzzz_zzzz_zzzz;
13         endcase
14
15 endmodule
16
```

- **parameter WORD = 16;** : 매개변수 WORD는 기본적으로 16으로 설정. 데이터의 비트 수를 의미
- **always @(*) case (sel)** : 입력 신호의 변화에 따라 동작하는 always 블록으로 'sel'의 값에 따라 'out'의 값을 결정
- **1'b0: out <= i0;** : 'sel'이 0일 때 'out'은 'i0'의 값
- **1'b1: out <= i1;** : 'sel'이 1일 때 'out'은 'i1'의 값
- **default: out <= 16'bzzzz_zzzz_zzzz_zzzz;** : 그 외의 경우 'out'은 불확정 상태로 설정

- **Memory**

  : 내부에 register file을 보유하고 있어 값의 저장이 가능하다. address입력을 받아 rnw신호에 따라 값을 저장할지 읽을지 결정하며 control logic에 의해 제어된다.

```
1  module memory_32x16(in_data, addr, clk, memrq, rw,
2                out_data);
3
4      input [15:0] in_data;
5      input [11:0] addr;
6      input clk, memrq, rw;
7      output reg [15:0] out_data;
8
9      reg [15:0] memory [0:31];
10     reg r_state;
11
12     always @(posedge clk) begin
13         if (memrq)
14             if (rw)
15                 r_state = 1;
16             else begin
17                 r_state = 0;
18                 memory[addr] = in_data;
19             end
20     end
21
22     always @(*)
23         if (memrq)
24             if (rw) out_data = memory[addr];
25
26 endmodule
27
```

- **reg [15:0] memory [0:31];** : 32개의 16비트 메모리를 정의
- **reg r_state;** : 1비트 레지스터
- **always @(posedge clk) begin**

  if (memrq) : 클록의 상승 에지에 의해 'memrq'가 활성화되면 동작
- **if(rw) r_state = 1;** : 'rw'가 1이면 읽기 상태로 설정
- **else begin r_state = 0;**

  memory[addr] = in_data; : 'rw'가 0이면 쓰기 상태로 설정, 'in_data'를 메모리의

'addr' 위치에 저장

- **Control logic**

  : 모든 register들의 동작을 제어하며, Fetch로부터 opcode 신호를 입력받아 해당 명령어에 대해 수행할 수 있도록 register들을 조절하고 내부의 execute/fetch state를 변화해가며 명령어 수행과 다음 명령어를 받아오는 동작을 하도록 한다.

```
module control_logic(in_opcode, acc_15, accz, clk, rst_n, a_sel, b_sel,
                     decode_ce, fetch_ce, acc_ce, acc_oe, execute_fs, rnw, memrq);

    input [3:0] in_opcode;
    input acc_15, accz, clk, rst_n;
    output reg [2:0] execute_fs;
    output reg a_sel, b_sel, decode_ce, fetch_ce, acc_ce, acc_oe;
    output reg rnw, memrq;

    reg ft, n_ft;
    reg [6:0] p_state; // 7'b_0000_000

    always @(*)
            if (rst_n == 1) p_state <= {in_opcode, ft, accz, acc_15};

    always @(*) begin
        if (rst_n == 1)
        case (in_opcode)
            4'b_0000: if (ft == 0) n_ft <= 1; else n_ft <= 0;
            4'b_0001: if (ft == 0) n_ft <= 1; else n_ft <= 0;
            4'b_0010: if (ft == 0) n_ft <= 1; else n_ft <= 0;
            4'b_0011: if (ft == 0) n_ft <= 1; else n_ft <= 0;
            4'b_0100: n_ft <= 0;
            4'b_0101: n_ft <= 0;
            4'b_0110: n_ft <= 0;
            4'b_0111: n_ft <= 0;
            default:  n_ft <= 1'b_x;
        endcase
    end

    always @(posedge clk) begin
        if (~rst_n) begin
            ft <= 1'b_0; //n_ft <= 1'b_0;
        end
        else
            ft <= n_ft;
    end

```

```
always @(p_state or rst_n) begin
    if (~rst_n) begin
        {a_sel, b_sel, acc_ce, decode_ce, fetch_ce, acc_oe, memrq, rnw} <= {8'b_0_0_1_1_1_0__1_1}; execute_fs <= 0;
    end
    else begin
        casex (p_state)
            //LDA S
            7'b_0000_0xx: begin {a_sel, b_sel, acc_ce, decode_ce, fetch_ce, acc_oe, memrq, rnw} <= {8'b_1_1_1_0_0_0__1_1}; execute_fs <= 3; end
            7'b_0000_1xx: begin {a_sel, b_sel, acc_ce, decode_ce, fetch_ce, acc_oe, memrq, rnw} <= {8'b_0_0_0_1_1_0__1_1}; execute_fs <= 4; end
            //STO S
            7'b_0001_0xx: begin {a_sel, b_sel, acc_ce, decode_ce, fetch_ce, acc_oe, memrq, rnw} <= {8'b_1_x_0_0_0_1__1_0}; execute_fs <= 3'bx; end
            7'b_0001_1xx: begin {a_sel, b_sel, acc_ce, decode_ce, fetch_ce, acc_oe, memrq, rnw} <= {8'b_0_0_0_1_1_0__1_1}; execute_fs <= 4; end
            //ADD S
            7'b_0010_0xx: begin {a_sel, b_sel, acc_ce, decode_ce, fetch_ce, acc_oe, memrq, rnw} <= {8'b_1_1_1_0_0_0__1_1}; execute_fs <= 1; end
            7'b_0010_1xx: begin {a_sel, b_sel, acc_ce, decode_ce, fetch_ce, acc_oe, memrq, rnw} <= {8'b_0_0_0_1_1_0__1_1}; execute_fs <= 4; end
            //SUB S
            7'b_0011_0xx: begin {a_sel, b_sel, acc_ce, decode_ce, fetch_ce, acc_oe, memrq, rnw} <= {8'b_1_1_1_0_0_0__1_1}; execute_fs <= 2; end
            7'b_0011_1xx: begin {a_sel, b_sel, acc_ce, decode_ce, fetch_ce, acc_oe, memrq, rnw} <= {8'b_0_0_0_1_1_0__1_1}; execute_fs <= 4; end
            //JMP S
            7'b_0100_xxx: begin {a_sel, b_sel, acc_ce, decode_ce, fetch_ce, acc_oe, memrq, rnw} <= {8'b_1_0_0_1_1_0__1_1}; execute_fs <= 4; end
            //JGE S
            7'b_0101_xx0: begin {a_sel, b_sel, acc_ce, decode_ce, fetch_ce, acc_oe, memrq, rnw} <= {8'b_1_0_0_1_1_0__1_1}; execute_fs <= 4; end
            7'b_0101_xx1: begin {a_sel, b_sel, acc_ce, decode_ce, fetch_ce, acc_oe, memrq, rnw} <= {8'b_0_0_0_1_1_0__1_1}; execute_fs <= 4; end
            //JNE S
            7'b_0110_x0x: begin {a_sel, b_sel, acc_ce, decode_ce, fetch_ce, acc_oe, memrq, rnw} <= {8'b_1_0_0_1_1_0__1_1}; execute_fs <= 4; end
            7'b_0110_x1x: begin {a_sel, b_sel, acc_ce, decode_ce, fetch_ce, acc_oe, memrq, rnw} <= {8'b_0_0_0_1_1_0__1_1}; execute_fs <= 4; end
            //STOP
            7'b_0111_xxx: begin {a_sel, b_sel, acc_ce, decode_ce, fetch_ce, acc_oe, memrq, rnw} <= {8'b_1_x_0_0_0_0__0_1}; execute_fs <= 3'bx; end
            default: begin {a_sel, b_sel, acc_ce, decode_ce, fetch_ce, acc_oe, memrq, rnw} <= {8'b_x_z_x_z_x_z__x_1}; execute_fs <= 3'd_7; end
        endcase
    end
end
```

- **reg ft, n_ft;** : 1비트 내부 레지스터
- **reg [6:0] p_state; // *7'b_0000_000*** : 'p_state'는 7비트 내부 레지스터입니다. 초기값은 7'b_0000_000
- **always @(*)**

  if(rst_n == 1) p_state <= {in_opcode, ft, accz, acc_15};

  : 리셋 상태에서 'p_state'를 갱신
- **always @(*) begin**

  if(rst_n == 1) : 리셋 상태에서 'n_ft'를 설정
- **case (in_opcode)**

  4'b_0000~0011: if(ft == 0) n_ft <= 1; else n_ft <= 0; : opcode가 0000에서 0011까지는 'ft' 상태에 따라 'n_ft'를 1 또는 0으로 설정

  4'b_0100: n_ft <= 0; : opcode가 0100에서 0111까지는 'n_ft'를 0으로 설정

  default: n_ft <= 1'b_x; : 그 외의 경우 'n_ft'는 정의되지 않은 값으로 설정

- **if(~rst_n) begin**

  ft <= 1'b_0; //n_ft <= 1'b_0; : 클록의 상승 에지에서 'ft'를 'n_ft'로 갱신

- **always @(p_state or rst_n) begin** : p_state나 rst_n의 변화에 따라 출력 신호를 갱신
- **if (~rst_n)begin {a_sel, b_sel, acc_ce, decode_ce, fetch_ce, acc_oe, memrq, rnw} <= {8'b_0_0_1_1_1_0__1_1}; execute_fs <= 0;** : 리셋 상태에서는 모든 출력을 초기화
- **casex (p_state)** : 값에 따라 출력 신호와 'execute_fs'를 설정
- **default: begin endcase** : 기본 경우는 모든 출력을 불확정 상태로 설정

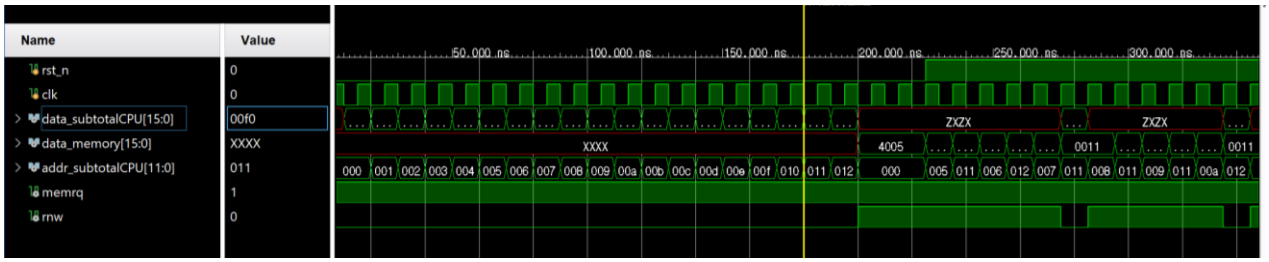
1.4 시뮬레이션 결과 및 분석

ㄱ. Testbench 설계

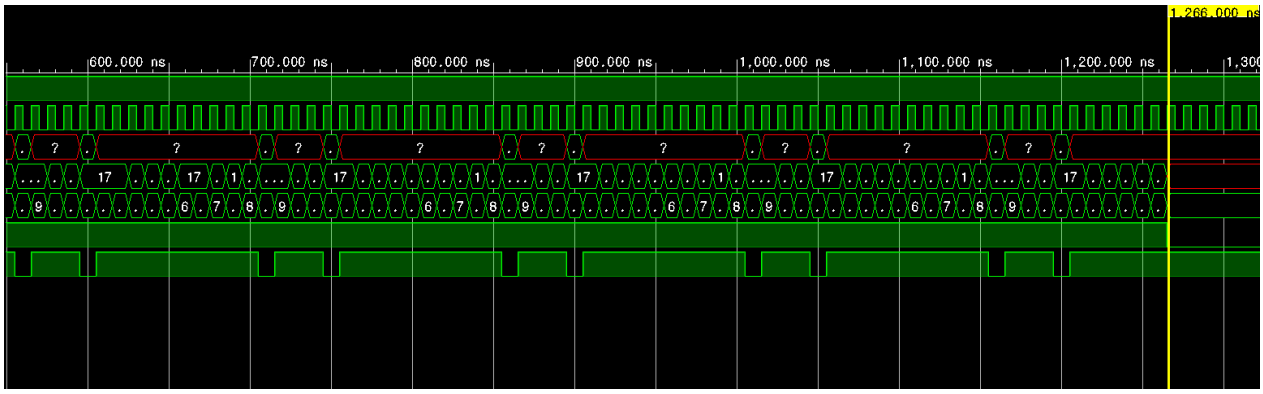
: 1.2에서 정의한 명령어의 정상적인 작동과 수열의 합($\sum_{i=S}^{N} i$ (N>=S))연산 task를 수행하기 위한 testbench를 설계한다. S가 15이고 N이 21이므로 15+16+17+18+19+20+21 = 126의 결과값이 도출되어야 하며, 처음 15를 sum을 대입하는 연산 외에 16부터 21까지 더하는 6번의 loop가 실행되어야 할 것이다.

```
`timescale 1 ns/100ps
module subtotalCPU_tb();

    reg rst_n, clk;
    wire [15:0] data_subtotalCPU, data_memory;
    wire [11:0] addr_subtotalCPU;
    wire memrq, rnw;

    subtotalCPU u_subtotalCPU(.in_data(data_memory), .rst_n(rst_n), .clk(clk),
          .out_data(data_subtotalCPU), .out_address(addr_subtotalCPU), .memrq(memrq), .rnw(rnw));
    memory_32x16 u_memory(.in_data(data_subtotalCPU), .addr(addr_subtotalCPU), .clk(clk), .memrq(memrq), .rw(rnw),
            .out_data(data_memory));

    initial begin
        clk <= 0; rst_n <= 0; force u_memory.rw = 0; force u_memory.memrq = 1;
        #10 force u_memory.addr =  0; force u_memory.in_data = 16'b_0000_0000_0001_0011; // LDA S    m[19]
        #10 force u_memory.addr =  1; force u_memory.in_data = 16'b_0001_0000_0001_0010; // STO sum  m[18]
        #10 force u_memory.addr =  2; force u_memory.in_data = 16'b_0001_0000_0001_0001; // STO i    m[17]
        #10 force u_memory.addr =  3; force u_memory.in_data = 16'b_0011_0000_0001_0000; // SUB N    m[16]
        #10 force u_memory.addr =  4; force u_memory.in_data = 16'b_0110_0000_0000_0110; // JNE loop1
        #10 force u_memory.addr =  5; force u_memory.in_data = 16'b_0111_0000_0000_0000; // STP
        #10 force u_memory.addr =  6; force u_memory.in_data = 16'b_0000_0000_0001_0001; // LDA i m[17] --- loop1 ----
        #10 force u_memory.addr =  7; force u_memory.in_data = 16'b_0010_0000_0001_0100; // ADD v1   m[20]
        #10 force u_memory.addr =  8; force u_memory.in_data = 16'b_0001_0000_0001_0001; // STO i    m[17]
        #10 force u_memory.addr =  9; force u_memory.in_data = 16'b_0010_0000_0001_0010; // ADD sum  m[18]
        #10 force u_memory.addr = 10; force u_memory.in_data = 16'b_0001_0000_0001_0010; // STO sum  m[18]
        #10 force u_memory.addr = 11; force u_memory.in_data = 16'b_0000_0000_0001_0001; // LDA i    m[17]
        #10 force u_memory.addr = 12; force u_memory.in_data = 16'b_0011_0000_0001_0000; // SUB N    m[16]
        #10 force u_memory.addr = 13; force u_memory.in_data = 16'b_0110_0000_0000_0110; // JNE loop1
        #10 force u_memory.addr = 14; force u_memory.in_data = 16'b_0111_0000_0000_0000; // STP
        #10 force u_memory.addr = 15; force u_memory.in_data = 16'b_xxxx_xxxx_xxxx_xxxx; // ---- additional data ----
        #10 force u_memory.addr = 16; force u_memory.in_data = 16'b_0000_0000_0001_0101; // N    m[16] = 21
        #10 force u_memory.addr = 17; force u_memory.in_data = 16'b_0000_0000_0000_0000; // i    m[17]
        #10 force u_memory.addr = 18; force u_memory.in_data = 16'b_0000_0000_0000_0000; // sum  m[18]
        #10 force u_memory.addr = 19; force u_memory.in_data = 16'b_0000_0000_0000_1111; // S    m[19] = 15
        #10 force u_memory.addr = 20; force u_memory.in_data = 16'b_0000_0000_0000_0001; // v1   m[20]
        #10 release u_memory.memrq; release u_memory.addr; release u_memory.in_data;release u_memory.rw;
        #25 rst_n <= 1;

    end

    always
        #5 clk <= ~clk;

endmodule
```

ㄴ. Simulation 파형

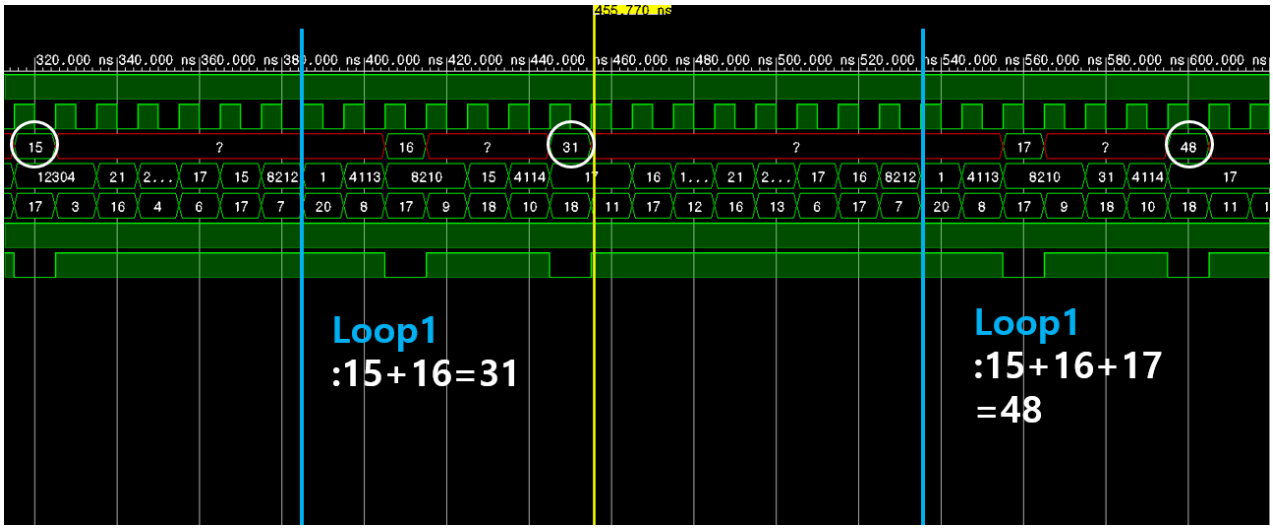


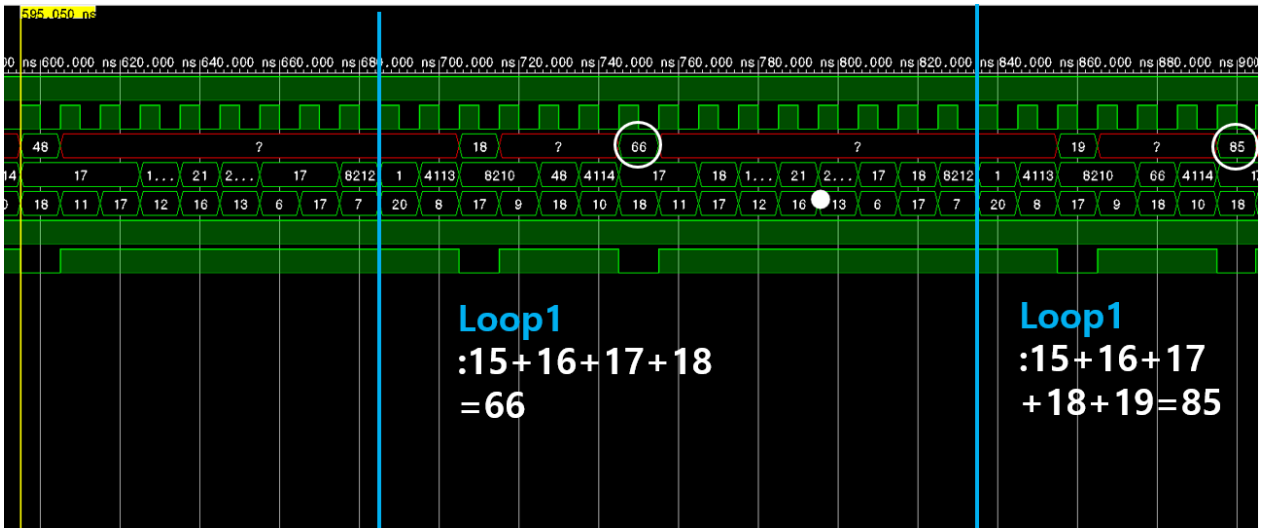
시뮬레이션 전체 파형은 다음과 같다.

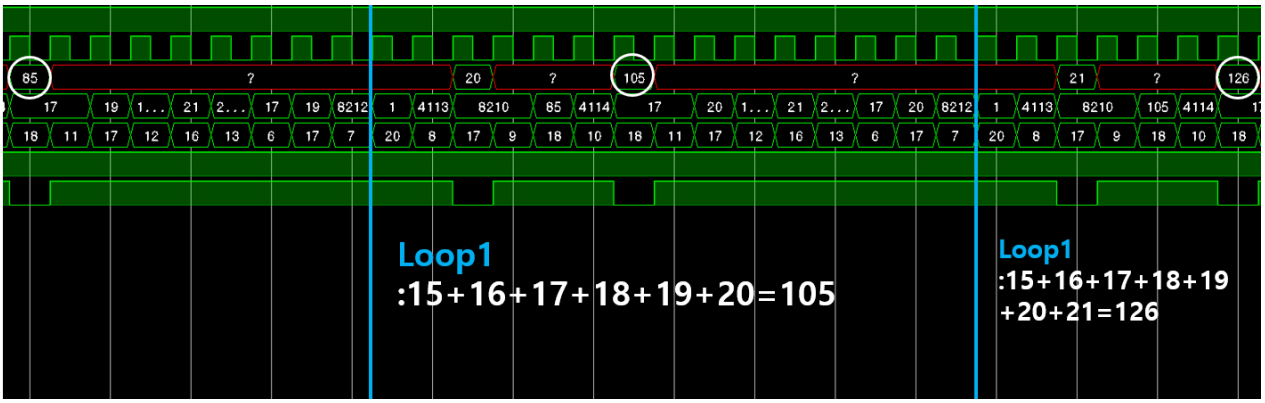


ㄷ. 파형 분석

: S가 15이고 N이 21이므로, sum에 15를 더하고 i(15)가 N(21)보다 작을 경우, loop1을 반복하기 때문에 i가 16, 17, 18, 19, 20, 21 총 6번의 loop1를 수행하고 프로그램이 종료될 것이다. 아래 첨부한 파형 사진을 통해 15부터의 수열의 합을 통해 총 6번의 loop1을 거쳐 126의 값을 산출한 것을 확인 할 수 있다.



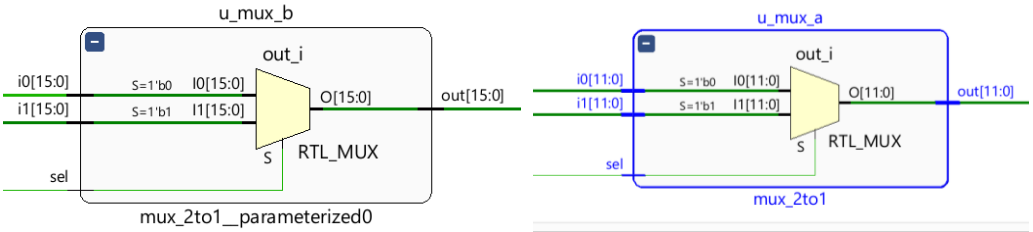

Loop1
:15+16+17+18
=66
Loop1
:15+16+17
+18+19=85


Loop1
:15+16+17+18+19+20=105
Loop1
:15+16+17+18+19
+20+21=126

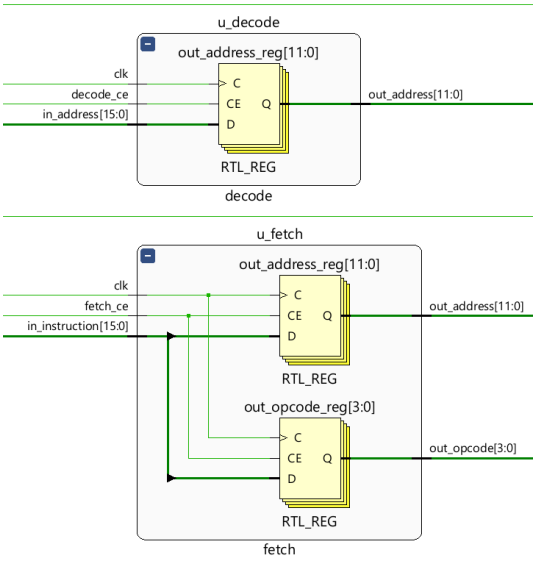
1.5 Synthesis

ㄱ. 각 모듈의 블록도 확인

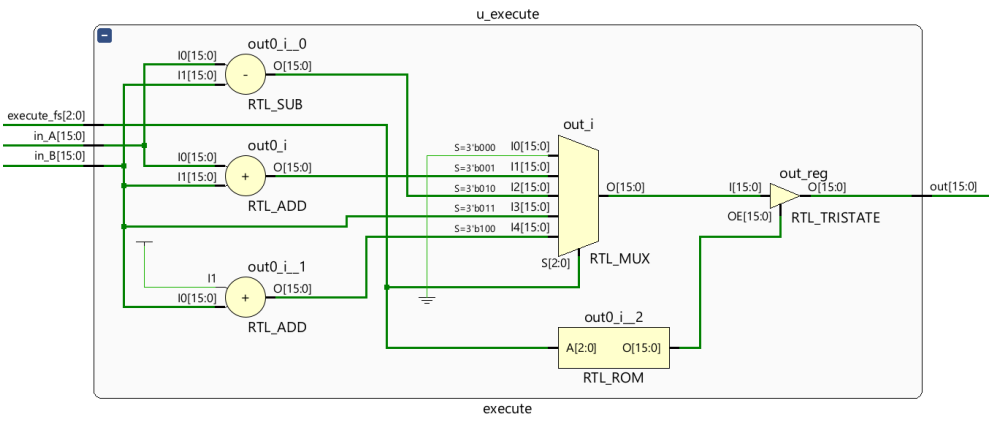
- **mux**



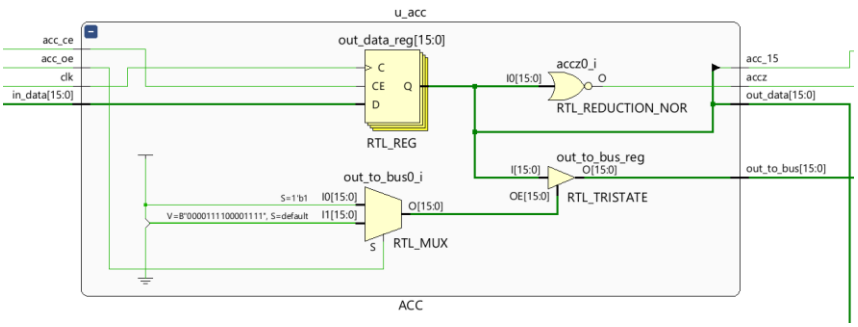
- **Fetch와 decode**



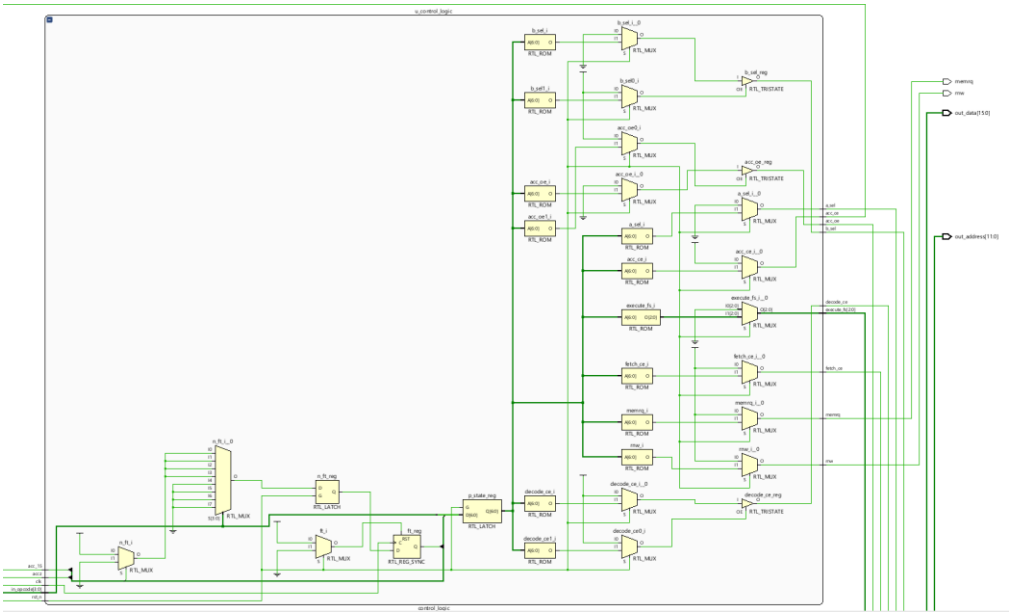
- **Execute**



- **ACC**

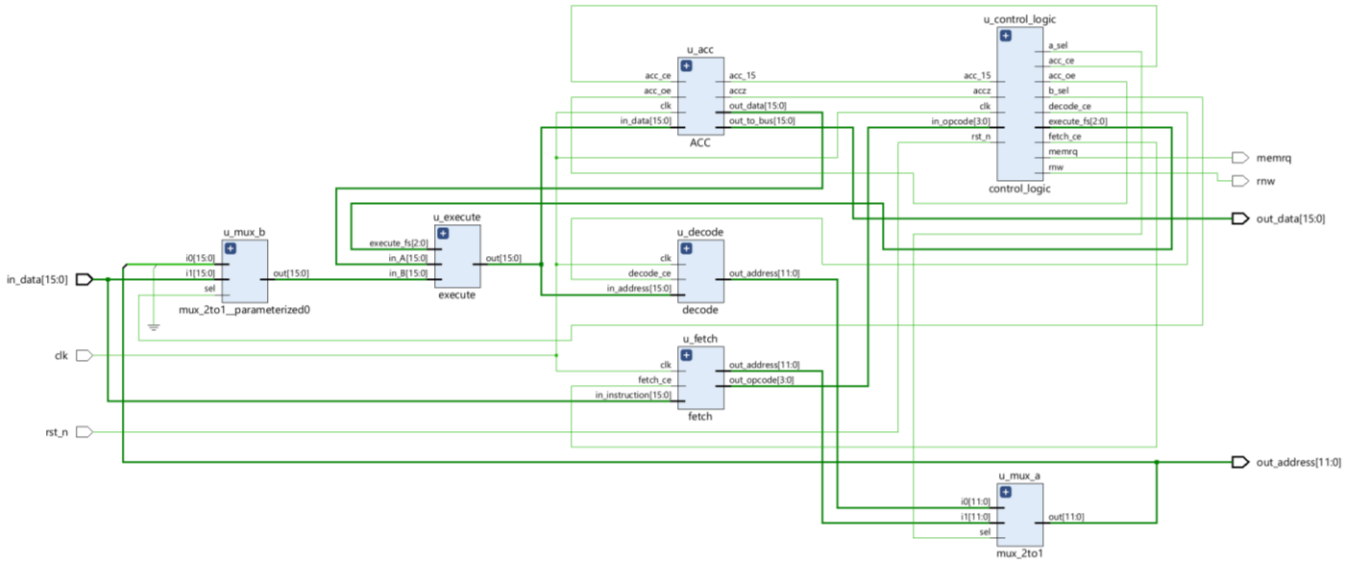


- **Control_logic**



ㄴ. Synthesis 진행 (schemetic)

본 프로젝트에서 설계한 CPU의 각 모듈을 vivado를 통해 합성을 진행하였다. 합성한 결과 다음사진과 같은 schemetic을 보인다. 2to1_mux 2개, fetch, decode, execute, ACC 그리고 control_logic 모듈이 합성되어 서로 연결되어 있는 모습을 확인할 수 있다.



2. 개발 과정 및 시행착오

2.1 Fetch- Decode – Execution CPU

: 8주차 과제를 해결하기 위하여 덧셈 기능이 있는 간단한 CPU(Fetch-Decode-Execution)를 제작하였다.

ㄱ. Memory

```
module Memory (
    input [7:0] addr,
    output [7:0] data
);
    reg [7:0] mem [0:255];

    initial begin
        // Load program
        mem[0] = 8'h01;  // LOAD 3
        mem[1] = 8'h02;  // ADD 9
        mem[2] = 8'hFF;  // HALT
    end

    assign data = mem[addr];
endmodule
```

8bit의 입력과 출력을 갖는 메모리를 제작하였다. 주소 0번지에는 01, 1번지에는 02, 2번지에는 FF를 저장하였고, 주소 addr에 해당하는 메모리 위치의 데이터를 data 출력에 할당하였다.

ㄴ. Fetch

```
module Fetch (
    input clk,
    input reset,
    output reg [7:0] pc,
    output [7:0] ir
);
    wire [7:0] memory_data;

    Memory memory (
        .addr(pc),
        .data(memory_data)
    );

    initial begin
        pc = 8'b0;
    end

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            pc <= 8'b0;
        end else begin
            pc <= pc + 1;
        end
    end

    assign ir = memory_data;
endmodule
```

Fetch 모듈에서는 명령어를 IR에 저장하고, PC를 관리하였다. 먼저, Memory에서 불러온 데이터를 wire에 저장하고 IR에 할당하였다. 이후 PC를 0으로 초기화하고, 클럭 신호에 따라 PC를 1씩 증가시켰다.

ㄷ. Decode

```
module Decode (
    input [7:0] ir,
    output reg [7:0] alu_op1,
    output reg [7:0] alu_op2,
    output reg [2:0] alu_operation,
    input [7:0] acc
);
    always @(*) begin
        case (ir)
            8'h01: begin
                alu_op1 = 8'd3;  // LOAD 3
                alu_op2 = 8'd0;
                alu_operation = 3'b000;  // ADD operation
            end
            8'h02: begin
                alu_op1 = acc;
                alu_op2 = 8'd9;
                alu_operation = 3'b000;  // ADD operation
            end
            default: begin
                alu_op1 = 8'd0;
                alu_op2 = 8'd0;
                alu_operation = 3'b000;  // NOP
            end
        endcase
    end
endmodule
```

Decode 모듈에서는 IR에서 제공하는 명령어를 해석하고, ALU에서 수행할 연산에 필요한 입력값을 설정하였다. alu_op1은 첫 번째 피연산자로 사용되는 8bit 값, alu_op2는 두 번째 피연산자로 사용되는 8bit 값이며, alu_operation은 ALU에서 수행할 연산을 지정하는 3bit 값이다. 명령어 '01'에 대해서는 alu_op1을 3, alu_op2를 0으로 지정하였고, ALU에서 add 연산을 수행하도록 하였다. 명령어 '02'에 대해서는 alu_op1을 acc의 값으로 설정하고, alu_op2를 9로 지정하였고, ALU에서 add 연산을 수행하도록 하였다. 이 외의 명령어에 대해서는 피연산자를 0으로 설정하고 아무 연산도 수행하지 않도록 하였다.

ㄹ. Execute

```
module Execute (
    input clk,
    input reset,
    input [7:0] alu_op1,
    input [7:0] alu_op2,
    input [2:0] alu_operation,
    output reg [7:0] acc,
    output [7:0] alu_result
);
    ALU alu (
        .op1(alu_op1),
        .op2(alu_op2),
        .operation(alu_operation),
        .result(alu_result)
    );

    initial begin
        acc = 8'b0;
    end

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            acc <= 8'b0;
        end else begin
            acc <= alu_result;
        end
    end
endmodule
```

Execute 모듈에서는 ALU를 사용하여 연산을 수행하고 결과를 acc에 저장하였다. initial begin과 (reset) begin을 통해 시뮬레이션이 시작될 때 acc를 0으로 초기화하였고, 클럭이 상승할 때마다 ALU연산 결과를 acc에 저장하도록 하였다.

ㅁ. ALU

```
module ALU (
    input [7:0] op1,
    input [7:0] op2,
    input [2:0] operation,
    output reg [7:0] result
);
    always @(*) begin
        case (operation)
            3'b000: result = op1 + op2; // ADD
            default: result = 8'b0;     // NOP
        endcase
    end
endmodule
```

ALU 모듈에서는 operation 값이 000, 즉 연산 명령일 경우 op1과 op2를 더한 값을 result에 저장하여 add 연산을 진행하였다.

ㅂ. CPU

```
module CPU (
    input clk,
    input reset,
    output [7:0] acc,  // Accumulator
    output [7:0] pc_out,
    output [7:0] ir_out,
    output [7:0] alu_op1_out,
    output [7:0] alu_op2_out,
    output [2:0] alu_operation_out,
    output [7:0] alu_result_out
);
    wire [7:0] pc;
    wire [7:0] ir;
    wire [7:0] alu_op1;
    wire [7:0] alu_op2;
    wire [2:0] alu_operation;
    wire [7:0] alu_result;

    // Fetch Stage
    Fetch fetch (
        .clk(clk),
        .reset(reset),
        .pc(pc),
        .ir(ir)
    );

    // Decode Stage
    Decode decode (
        .ir(ir),
        .alu_op1(alu_op1),
        .alu_op2(alu_op2),
        .alu_operation(alu_operation),
        .acc(acc)
    );

    // Execute Stage
    Execute execute (
        .clk(clk),
        .reset(reset),
        .alu_op1(alu_op1),
        .alu_op2(alu_op2),
        .alu_operation(alu_operation),
        .acc(acc),
        .alu_result(alu_result)
    );

    assign pc_out = pc;
    assign ir_out = ir;
    assign alu_op1_out = alu_op1;
    assign alu_op2_out = alu_op2;
    assign alu_operation_out = alu_operation;
    assign alu_result_out = alu_result;
endmodule
```

CPU 모듈에서는 Fetch, Decode, Execute 세 단계를 포함하였다. 여러 wire을 선언하여 각 단계 간 데이터 전달을 용이하게 하였고, 내부 신호를 외부 출력으로 연결하여 파형을 모니터링할 수 있도록 하였다.

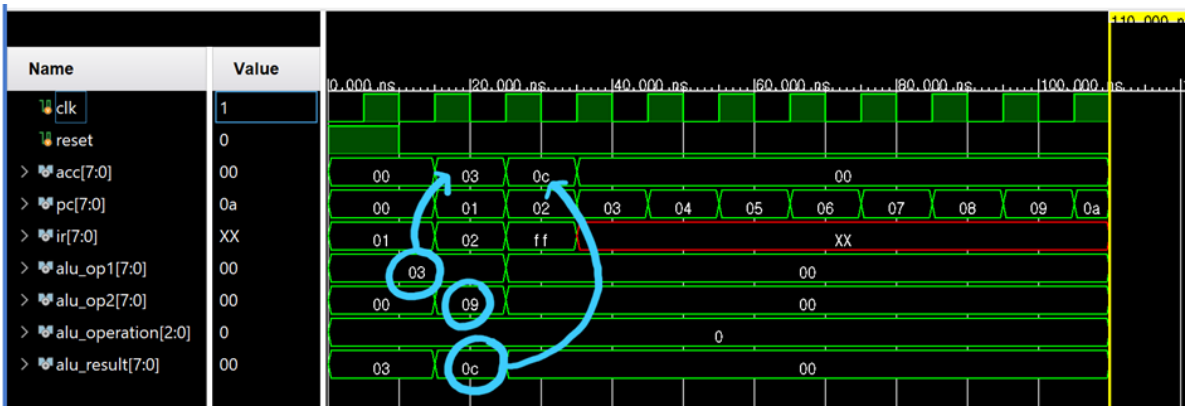
ㅅ. Testbench

:Testbench는 챗gpt의 도움을 받아 다음과 같이 작성하였다.

```
module CPU_tb;
    reg clk;
    reg reset;
    wire [7:0] acc;
    wire [7:0] pc;
    wire [7:0] ir;
    wire [7:0] alu_op1;
    wire [7:0] alu_op2;
    wire [2:0] alu_operation;
    wire [7:0] alu_result;

    CPU cpu (
        .clk(clk),
        .reset(reset),
        .acc(acc),
        .pc_out(pc),
        .ir_out(ir),
        .alu_op1_out(alu_op1),
        .alu_op2_out(alu_op2),
        .alu_operation_out(alu_operation),
        .alu_result_out(alu_result)
    );

    initial begin
        clk = 0;
        forever #5 clk = ~clk;  // 10 ns period clock
    end

    initial begin
        reset = 1;
        #10;
        reset = 0;
        #100;  // Run for 100 time units

        $finish;
    end

    initial begin
        $monitor("Time: %0t | PC: %0d | IR: %0d | Acc: %0d | ALU_OP1: %0d | ALU_OP2: %0d | ALU_OP: %0d | ALU_RESULT: %0d",
                 $time, pc, ir, acc, alu_op1, alu_op2, alu_operation, alu_result);
    end
endmodule

```

ㅇ. Waveform

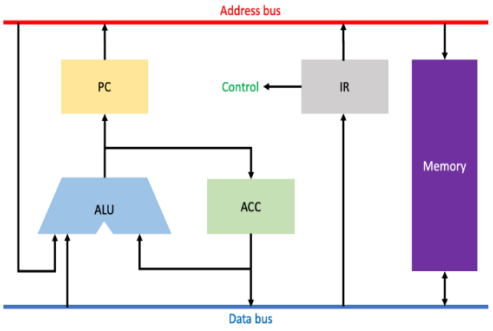
파형을 통해 CPU의 동작 과정을 보았을 때, 0ns에 op1에 3이 할당된 후 acc에 저장되었다. 20ns에는 op2에 9가 할당되었고 acc에 저장된 3을 불러와 연산을 진행하였다. 결과값 0c(12)의 값을 acc에 저장였고 alu_result에서 그 값을 확인할 수 있었다..



2.2 MU0

프로젝트를 진행하기 위해 다양한 프로세서에 대해 찾아보던 중 MU0를 접하게 되었다. MU0는 맨체스터 대학에서 설계한 교육용 프로세서로, 가장 간단한 프로세서이기 때문에 프로그램 경험이 전혀 없는 팀원들이 실습 및 연구하기에 적합하다고 판단하였다. PC, ACC, ALU, IR이 기본적인 프로세서 요소이며, 앞서 제작한 CPU에 IR 모듈을 추가하여 구현해보았다.

Opcode	Instruction	Effect
0000	LDA s	ACC := mem_{16}[S]
0001	STO s	mem_{16}[S] := ACC
0010	ADD s	ACC := ACC +mem_{16}[S]
0011	SUB s	ACC := ACC - mem_{16}[S]
0100	JMP s	PC := S
0101	JGE s	if ACC >= 0 PC := S
0110	JNE s	if ACC != 0 PC := S
0111	STP	stop



ㄱ. Memory

```
module mem_model(
        input clk, memrq, rnw,
        input [11:0] addr,
        inout [15:0] data);

    localparam LDA = 4'b0000, STO = 4'b0001, ADD = 4'b0010,
               SUB = 4'b0011, JMP = 4'b0100, JGE = 4'b0101,
               JNE = 4'b0110, STP= 4'b0111;

    reg [15:0]mem[0:4095];


    assign data = memrq&rnw ? mem[addr] : 16'hz;

    always @ (negedge clk)
        if(memrq & !rnw) mem[addr] <= data;


    //if(a == b)
    //a = a + c
    //else
    //a = a - 2c
    initial begin

        mem[   0] = {LDA, 12'h64}; // 100
        mem[   1] = {SUB, 12'h65}; // 101
        mem[   2] = {JNE, 12'h6};  // 6
        mem[   3] = {LDA, 12'h64}; // 100
        mem[   4] = {ADD, 12'h65}; // 101
        mem[   5] = {JMP, 12'h9};  // 9
        mem[   6] = {LDA, 12'h64}; //100
        mem[   7] = {SUB, 12'h66}; //102
        mem[   8] = {SUB, 12'h66}; //102
        mem[   9] = {STO, 12'h64}; //100
        mem[  10] = {STP, 12'h0};  //0

        mem[ 100] = {LDA, 12'h444}; //1092
        mem[ 101] = {LDA, 12'h222}; //546
        mem[ 102] = {LDA, 12'h111}; //273
    end
```

: MU0의 각 명령어를 나타내는 4비트 상수와 메모리 공간을 정의한 후 데이터 버스를 연결하였다. 예제로는 a와 b를 비교하여 같으면 'a+c', 다르면 'a-2c' 를 수행하도록 하였다.

ㄴ. PC

```
module pc_reg(
    input clk,
    input pcce,
    input [15:0] alu,
    output reg [15:0] pc);

    wire  [15:0] pc1;
    assign pc1 = pc;

    always @ (negedge clk)
        if(pcce) pc <= alu;
        else pc <= pc1;

endmodule
```

Negative edge에서 clk신호를 트리거시켰다. 만약 PCce 신호가 참이면 PC는 ALU 값을 받도록 하였다.

ㄷ. ACC

```
module acc_reg (
    input clk,
    input accce,
    input [15:0] alu,
    output acc15,
    output accz,
    output reg [15:0] acc);

    wire [15:0] acc1;
    assign acc1 = acc;

    assign acc15 = acc[15];
    assign accz = (acc==16'h00) ? 1 : 0;

    always @ (negedge clk)
        if(accce) acc <= alu;
        else acc <= acc1;

endmodule
```

ACC의 값을 할당하고, ACC 레지스터의 값이 0이면 ACCz는 1, 그렇지 않으면 0을 출력하도록 하였다. ACCce 신호가 활성화되면 ALU 입력값을 ACC에 저장하고, 그렇지 않으면 acc1에 할당된 값을 유지한다.

ㄹ. IR

```
module ir_reg(
    input clk,
    input irce,
    input [15:0] data,
    output reg [15:0] ir);

    wire  [15:0] ir1;
    assign ir1 = ir;

    always @ (negedge clk)
        if(irce) ir <= data;
        else ir <= ir1;

endmodule
```

Data bus의 값을 저장하는 IR 레지스터를 설계하였다.

ㅁ. ALU

```
module alu_16bit(
    input reset,
    input [15:0] a,
    input [15:0] b,
    input [1:0] alufs,
    output reg [15:0] sum);

    //alufs[1:0]
    //2'b00 = b
    //2'b01 = b+1
    //2'b10 = a+b
    //2'b11 = a-b

    always @ (reset,a,b,alufs) begin
        if(reset)
            sum = 0;
        else begin
            case(alufs)
                2'b00 : sum = b;
                2'b01 : sum = b+1;
                2'b10 : sum = a+b;
                2'b11 : sum = a-b;
            endcase
        end
    end

endmodule
```

2bit의 신호를 통해 수행할 연산 4가지(덧셈 및 뺄셈)를 지정하였다.

ㅂ. MU0

```
module mu0(
    input reset, clk,
    inout [15:0] data,
    output [11:0] addr,
    output memrq, rnw);

    wire [15:0] alu, ir, pc, b, acc;
    wire [1:0] alufs;
    wire pcce, irce, asel, bsel, accoe;
    wire accce, acc15, accz;

    pc_reg pc_reg(.clk(clk), .pcce(pcce), .alu(alu), .pc(pc));
    ir_reg ir_reg(.clk(clk), .irce(irce), .data(data), .ir(ir));
    mux12bit mux12(.a(ir[11:0]), .b(pc[11:0]), .s(asel), .out(addr));
    mux16bit mux16(.a(data), .b({4'b0000, addr}), .s(bsel), .out(b));
    acc_reg acc_reg(.clk(clk), .accce(accce), .alu(alu), .acc(acc), .acc15(acc15), .accz(accz));
    tri16bit tri16(.in(acc), .oe(accoe), .out(data));
    alu_16bit alu_16bit(.reset(reset), .a(acc), .b(b), .alufs(alufs), .sum(alu));
    fsm fsm(.reset(reset), .clk(clk), .opcode(ir[15:12]), .accz(accz), .acc15(acc15),
            .asel(asel), .bsel(bsel), .accce(accce), .pcce(pcce), .irce(irce),
            .accoe(accoe), .alufs(alufs), .memrq(memrq), .rnw(rnw));
endmodule
```

위의 하위 모듈을 통합하고 인스턴스화 하였다.

ㅅ. 이 외의 모듈 코드

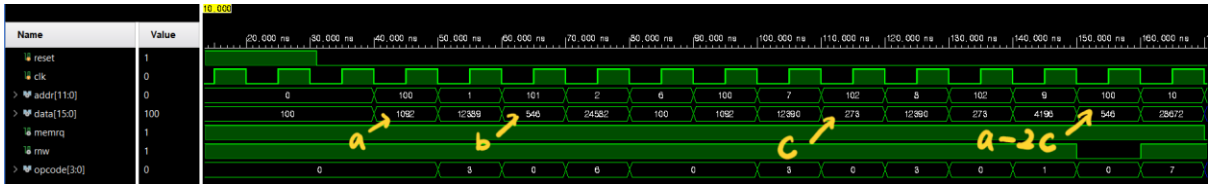
```
module mux12bit(
    input s,
    input [11:0] a,b,
    output [11:0] out);

    assign out = s? a:b;
endmodule
```

```
module mux16bit(
    input s,
    input [15:0] a,b,
    output [15:0] out);

    assign out = s? a:b;
endmodule
```

```
module tri16bit(
    input [15:0] in,
    input oe,
    output [15:0] out);

    assign out = oe? in : 16'hz;
endmodule
```

두 가지의 Mux를 이용하여 필요한 입력값을 선택하였다. Tri state buffer을 생성하여 ACC의 출력을 data bus에 싣도록 하였다.

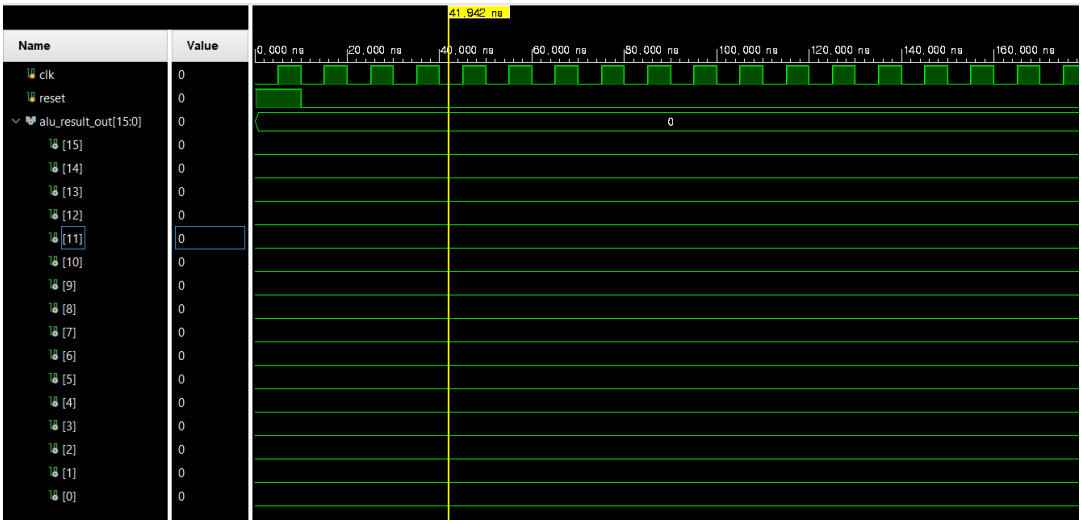
ㅇ. Waveform



메모리로부터 40ns에 1092라는 a값을, 60 ns에 546이라는 b값을, 110ns에 273이라는 c값을 불러왔다. a,b 값이 다르기 때문에 'a-2c'를 진행하였고, 1092-2*273인 546을 정상적으로 확인하였다. 또한 연산 외에도 데이터를 불러오고 저장하는 등의 다른 명령어도 정상적으로 동작함을 확인하였다.

2.3 시행착오

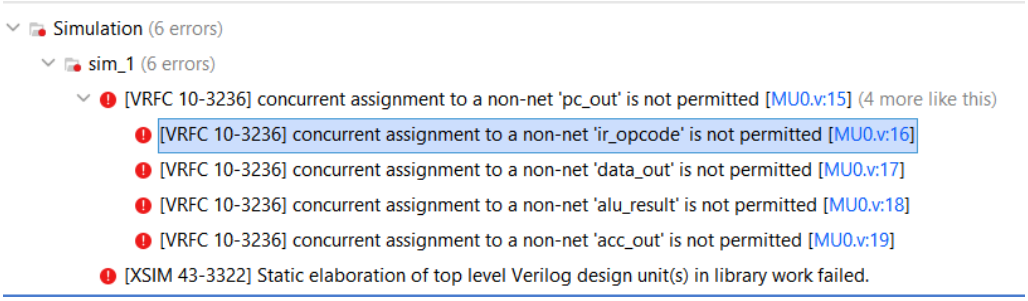
ㄱ. 시뮬레이션 오류 및 파형에 원하는 데이터가 출력되지 않음



&display 명령어를 도입하여 해결하고, display 명령어에 대해 공부하였다. 문법적 오류로 인한 시뮬레이션 오류가 발생했음을 알게 되어 최종 프로젝트를 할 때 여러 번의 검토를 진행하였다.

ㄴ. instance 오류

다양한 변수와 모듈을 연결하는 과정에서 오류가 발생하였다. 각 모듈의 코드에서 사용된 입, 출력값과 명령어를 할당하는 코드를 수정하여 해결하였다.

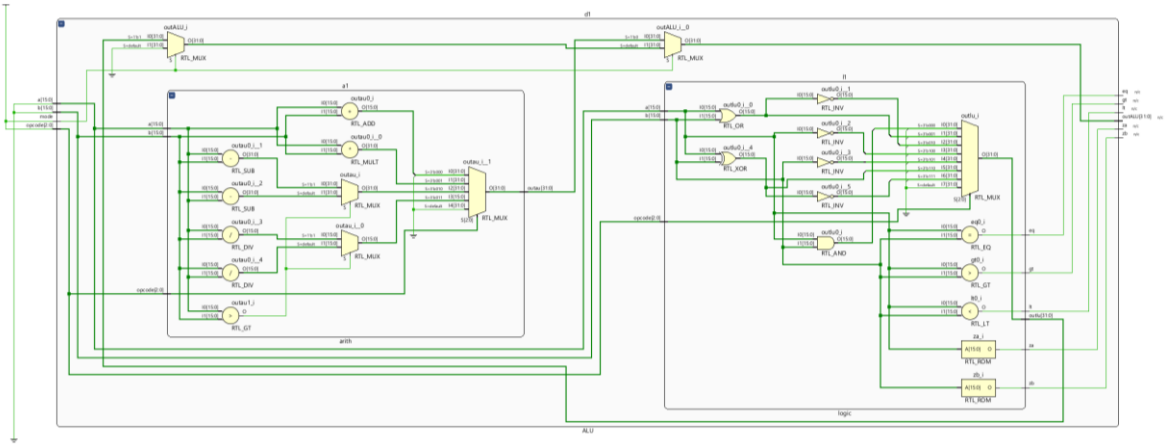


ㄷ. 코드 작성 중의 문법적 오류

강의록에 첨부된 코드와 챗지피티를 활용하여 수정하였다.

3. 향후 개발 방향

다양한 cpu 설계의 실습 예시를 찾아보던 중, clk과 reset 신호를 사용하지 않은 sequential logic circuit의 schematic 이미지를 접하게 되었다.



✓ 순차회로에서 사용되는 clk 신호의 중요성과 비동기 순차회로에 대해 연구한다.

✓ 본 이미지의 순차회로를 구현한다. 이때, 클럭신호를 추가한 순차회로를 추가로 만들어 두 회로의 차이점을 연구한다.

4. 부록

4.1 Github 주소

1팀 git https://github.com/SSUminiCPU/1teamCPU

팀원 git

김동희 git https://github.com/DongH-e-e/Digital-logic-circuit

김형주 git https://github.com/gudwn1104/Digital-logic-circuit

정민지 git https://github.com/wjdalswl-airair/Digital-logic-circuit

4.2 회의록

ㄱ. 제1차 회의록(2024.04.15)

[1팀] 제 1차 회의록

북마크 링크복사

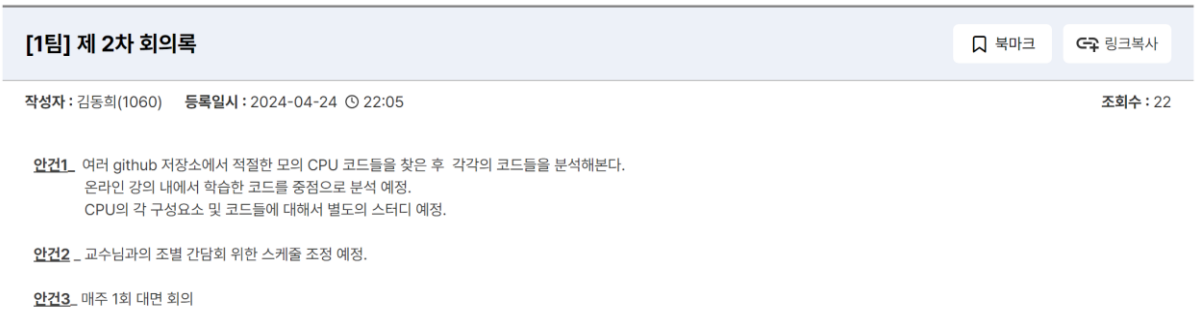
작성자 : 김동희(1060) **등록일시 :** 2024-04-15 22:09 **조회수 :** 48

안건 1. 역할 분담

김동희 : 서기 및 회의록 작성
김형주 : 팀장,전체 총괄 및 감독
정민지 : CTO

안건2. 수행 내용 (계획)
CPU의 기본 원리 및 작동 원리에 대해 알아보기

ㄴ. 제2차 회의록(2024.04.24)



[1팀] 제 2차 회의록

북마크 링크복사

작성자 : 김동희(1060) **등록일시 :** 2024-04-24 22:05 **조회수 :** 22

안건1_ 여러 github 저장소에서 적절한 모의 CPU 코드들을 찾은 후 각각의 코드들을 분석해본다.
온라인 강의 내에서 학습한 코드를 중점으로 분석 예정.
CPU의 각 구성요소 및 코드들에 대해서 별도의 스터디 예정.

안건2 _ 교수님과의 조별 간담회 위한 스케줄 조정 예정.

안건3_ 매주 1회 대면 회의

ㄷ. 제3차 회의록(2024.04.24)

[1팀] 제 3차 회의록

북마크 링크복사

작성자 : 김동희(1060) **등록일시 :** 2024-05-01 19:38 **조회수 :** 25

1_ CPU 기획
- 간단한 MU0 구조를 기반으로 CPU 설계 기획
- PC, ACC, ALU, IR, Memory 구성
- 4bit OP코드, 메모리 주소를 저장하는 12bit → 총 16비트의 RISC 포멧

2_회의 일정
- 차주 (월) zoom 미팅을 통해 구체적인 계획 수립 및 보고서 양식 결정

ㄹ. 제4차 회의록(2024.04.24)

[1팀] 제 4차 회의록 북마크 링크복사

작성자 : 김동희(1060) **등록일시 :** 2024-05-08 23:31 **조회수 :** 33

1. 간담회 장소 제안 _ 취향&가야성 / 광양불고기 / 황새골 / 팽여사버섯오리명가 등등

2. Mu0_프로세스 스터디
- 동작 원리
- 기본 구조 이해

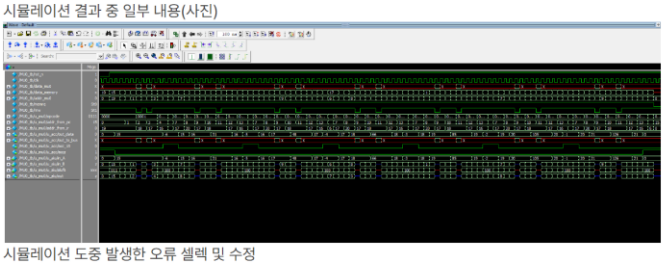
3. github 임의의 Mu0코드 시뮬레이션 돌려보기
- github 코드 시뮬레이션 돌리기
- 금주 (목) 대면 회의를 통해 수행 예정 _ 도서관 세미나실 예약

ㅁ. 제5차 회의록(2024.04.24)

[1팀] 제 5차 회의록 북마크 링크복사

작성자 : 김동희(1060) **등록일시 :** 2024-05-15 20:41 **조회수 :** 35

1. MU0 시뮬레이션
   - 시뮬레이션 결과 중 일부 내용(사진)



   - 시뮬레이션 도중 발생한 오류 셀렉 및 수정

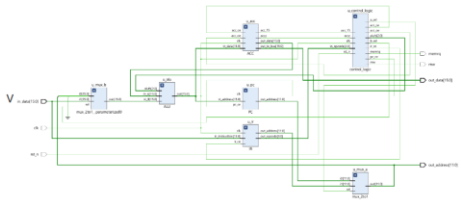
2. MU0 프로세서 세부 분석
   - ACC, ALU, IR, Memory, Mux(2to1), Control logic 등 MU0의 각 모듈 코드 세부 분석

ㅂ. 제6차 회의록(2024.04.24)

[1팀] 제 6차 회의록 북마크 링크복사

작성자 : 김동희(1060) **등록일시 :** 2024-05-22 23:49 **조회수 :** 13

1. MU0 schematic 중간 check



2. MU0 16bit 에서 32bit로 변환 계획
   - ALU 연산 추가, 기존의 4x1 Mux 모듈 등을 develop

3. 기존 MU0 processor에 추가할만한 새로운 모듈에 대한 고찰

ㅅ. 제7차 회의록(2024.04.24)

[1팀] 제 7차 회의록

북마크 링크복사

작성자 : 김동희(1060) **등록일시 :** 2024-05-29 22:30 **조회수 :** 13

1. 중간 보고서 작성
    - 1~6차 회의록 내용 & 설계 모듈 설명 & simulation결과 & 경험한 에러 사항 및 에러 분석 등의 내용 언급

2. CPU 프로젝트 최종 점검 - 6월 4일 대면 회의를 통해 진행 예정
    - 최종 보고서 작성
    - 설계 내용 최종 수정 및 완성
    - 발표 내용 정리

ㅇ. 제8차 회의록(2024.04.24)

[1팀] 제 8차 회의록

북마크 링크복사

작성자 : 김동희(1060) **등록일시 :** 2024-06-05 21:49 **조회수 :** 9

1. 프로젝트 최종 완성
    - 마지막 시뮬레이션 실행
    - synthesis 후 주파수 확인

2. 프로젝트 마무리하기
    - 김동희: 최종 보고서작성 _ 설계한 CPU 명령어&모듈&시뮬레이션 등의 내용 작성
    - 정민지: 최종 보고서작성_ 프로젝트 과정 중의 여러 시행착오 등의 내용 작성
    - 김형주: 발표ppt 제작 및 발표준비